Single pass graph sparsification in distributed stream processing systems*

Ashish Goel† Michael Kapralov‡ Olga Kapralova§ Sanjeev Khanna¶

November 29, 2011

Abstract

We give a distributed one pass streaming algorithm for graph sparsification. Besides producing a sparsifier, our algorithm maintains a hierarchy of UNION-FIND data structures in a distributed manner that efficiently support queries of strong connectivities between pairs of vertices. An important component of the algorithm is an implementation of UNION-FIND queries over an Active Distributed Hash Table that guarantees good load balancing properties. This is achieved via a single step of what is known in the literature as the zig-zag heuristic. We provide theoretical guarantees for the load balancing achieved by this heuristic, and show how the structure of our sparsification scheme ensures good load balancing across the hierarchy of UNION-FIND data structures maintained by the algorithm.

We also present simulation results on synthetic as well as real world data verifying the load balancing properties and the quality of approximation of strong connectivities achieved by the algorithm.

1 Introduction

Processing large graphs is an essential component of many core Web services, including search engines such as Google and social networks such as Twitter and Facebook. These underlying graphs are often too large to effectively process on a single computer. While there is growing understanding of offline graph algorithms on bulk-synchronous distributed systems such as Map-Reduce [6, 1] and Pregel [17], relatively little progress has been made towards real-time distributed implementations. In this paper, we present a distributed streaming algorithm for graph sparsification. We choose graph sparsification as the problem to study since it has emerged as a basic sub-routine for many graph operations over the last fifteen years, and is as canonical a part of the modern algorithmic graph theory tool-kit as breadth-first or depth-first search.

We are going to assume that the underlying computational substrate is an *Active DHT*. A DHT (Distributed Hash Table) is a distributed `<key, value>` store which allows Lookups, Inserts, and Deletes on the basis of the Key. The term Active refers to the fact that we assume that an arbitrary User Defined Function (UDF) can be locally executed on a `<key, value>` pair wherever it is stored, in addition to Insert, Delete, and Lookup. Yahoo's S4 [21] and Twitter's Storm [2] are two examples of (now open-sourced) Active DHTs which are gaining widespread use. The Active DHT model is broad enough to act as a distributed stream processing system and as a continuous version of Map-Reduce, and largely subsumes Pregel.

*A more technical exposition of the streaming version of refinement sampling is given in an unpublished manuscript [9]

†Departments of Management Science and Engineering and (by courtesy) Computer Science, Stanford University. Email: ashishg@stanford.edu. Research supported in part by NSF award IIS-0904325.

‡Institute for Computational and Mathematical Engineering, Stanford University. Email: kapralov@stanford.edu.

§Department of Electrical Engineering, University of California at Riverside. Email: okapralova@ee.ucr.edu

¶Department of Computer and Information Science, University of Pennsylvania, Philadelphia PA. Email: sanjeev@cis.upenn.edu. Supported in part by NSF Awards CCF-1116961 and IIS-0904314.

The notion of graph sparsification was introduced in [12] and further developed in [4], where the authors gave a near linear time procedure that takes as input an undirected graph G on n vertices and constructs a weighted subgraph H of G with $O(n \log n/\epsilon^2)$ edges such that the value of every cut in H is within a $1 \pm \epsilon$ factor of the value of the corresponding cut in G. This algorithm has subsequently been used to speed up algorithms for finding approximately minimum or sparsest cuts in graphs ([4, 15]), as well as in a host of other applications (e.g. [13]). A more general class of spectral sparsifiers was recently introduced by Spielman and Srivastava in [23]. The algorithms developed in [4] and [23] take near-linear time in the size of the graph and produce very high quality sparsifiers, but require random access to the edges of the input graph G, which is often prohibitively expensive in applications to modern massive data sets. The streaming model of computation, which restricts algorithms to use a small number of passes over the input and space polylogarithmic in the size of the input, has been studied extensively in various application domains (e.g. [19]), but has proven too restrictive for even the simplest graph algorithms (even testing $s - t$ connectivity requires $\Omega(n)$ space). The less restrictive semi-streaming model, in which the algorithm is restricted to use $\tilde{O}(n)$ space, is more suited for graph algorithms [7]. The problem of constructing graph sparsifiers in the semi-streaming model was recently considered by Anh and Guha [3], who gave a one-pass algorithm for finding Benczúr-Karger type sparsifiers with a slightly larger number of edges than the original Benczúr-Karger algorithm, i.e. $O(n \log n \log \frac{m}{n}/\epsilon^2)$. In [14], Kelner and Levin gave an extremely elegant one-pass streaming solution for the problem of constructing stronger *spectral* sparsifiers with $O(n \log n/\epsilon^2)$.

Apart from the issue of random access vs disk, the semi-streaming model is also important for scenarios where edges of the graph are revealed one at a time by an external process. For example, this application maps well to online social networks where edges arrive one by one, but efficient network computations may be required at any time, making it particularly useful to have a dynamically maintained sparsifier.

While it is known how to obtain sparsifiers of $O(n \log n/\epsilon^2)$ in the streaming setting, none of the known approaches seem to translate easily to the distributed setting, which is the main focus of this paper. We now discuss known approaches for computing sparsifiers, in particular, the known algorithms for estimating the sampling rates that are sufficient for sampling:

(a) Computing edge or vertex connectivities [4, 8], using the oracle of Nagamochi and Ibaraki [20]. In traditional main memory setting, this requires edges to be scanned in a very specific order (grouped by vertex, with vertices at higher "levels" scanned first). While this appears to rule out a streaming algorithm, one can get around this difficulty (e.g., [3]) by computing an intermediate representation of the sparsifier and then re-sparsifying. However, this re-sparsification approach also does not appear to translate to a distributed implementation;

(b) It was shown in [8, 10] that random spanning trees can be used to obtain sparsifiers. While these results are very elegant, the best known methods for generating uniformly random spanning trees currently require $\Omega(m\sqrt{n})$ time even in the random access model;

(c) Solving SDD Linear Systems [24] to approximate edge resistances. No distributed streaming algorithms are known for solving these systems; in fact our results could be a potential approach towards solving SDD linear systems using Active DHTs since recent fast algorithms for SDD linear systems [16] use spectral sparsifiers as a subroutine;

(d) Using random spanners to estimate proximity thresholds [11]. No distributed streaming method for constructing spanners is known.

Our results and techniques In this paper we present a simple sparsifier based on the UNION-FIND data-structure. This sparsifier is amenable to streaming implementation on an active DHT. We first slightly modify the UNION-FIND data structure so that for m edge arrivals over n nodes

1. The total number of network calls and total processing time are $O((m + n) \log^* n)$

2. Each key-value pair is of size $O(1)$, the key-value pairs can be striped across machines using an arbitrary hash function, and the total number of key-value pairs is $O(n)$

3. The maximum number of references (reads + writes) to any key-value pair is $O(n)$

4. Any one edge insertion incurs at most $O(\log n)$ processing time and network calls.

We then show how to adapt this UNION-FIND data structure to produce combinatorial sparsifiers in a single pass over the data in a distributed setting using what we refer to as the *refinement sampling scheme*. At a high level, the basic idea is to sample edges at geometrically decreasing rates, using the sampled edges at each rate to refine the connected components from the previous rate. The sampling rate at which the two endpoints of an edge get separated into different connected components is used as an approximate measure of the "strength" of that edge. We use refinement sampling to obtain two algorithms for computing Benczúr-Karger type sparsifiers of undirected graphs in a distributed setting.

The first algorithm requires $O(\log n)$ passes, $O(\log n)$ space per node, $O(\log^2 n \log^* n)$ work per edge and produces sparsifiers with $O(n \log^2 n/\epsilon^2)$ edges. The second algorithm requires a single pass over the edges of the graph, $O(\log^2 n)$ space per node, $O(\log^2 n \log^* n)$ work per edge and produces sparsifiers with $O(n \log^3 n/\epsilon^2)$ edges.

We present results of experimental evaluation of the load balancing properties and the quality of approximation of edge strengths on both synthetic and real world data. The results confirm favorable load balancing properties of the algorithm, and show that the algorithm obtains a small constant factor approximation to strengths of most edges both in the multi-pass and single pass settings.

Related work To the best of our knowledge, our algorithm is the first distributed implementation of graph sparsification. The zig-zag, or immediate parent check heuristic was introduced in [22] and evaluated experimentally in [18]. However, no theoretical guarantees on its performance have been given.

Organization In section 2 we discuss the distributed implementation of the UNION-FIND data structure and prove that it possesses good load balancing properties. In section 3 we review the definitions relevant to graph sparsification, and in section 4 we present the basic refinement sampling scheme, and show how it implies an $O(\log n)$-pass sparsification algorithm. In section 5 we extend the refinement scheme to yield a sparsifier in one-pass, and finally in section 6 we present experimental evaluation of the algorithm on synthetic as well as real-world data.

2 Distributed UNION-FIND and load balancing

In this section we describe a distributed implementation of the UNION-FIND data structure and prove its load-balancing properties. We start by recalling the definition of UNION-FIND and its basic properties. The UNION-FIND data structure is used to maintain a partition of a universe $\mathcal{U}$ of size n into disjoint sets. The data structure supports two queries: FIND(u) which for $u \in \mathcal{U}$ outputs a designated representative element of the set that u belongs to, and UNION(u, v), which merges the two sets that u and v belong to. The classical UNION-FIND data structure maintains a forest of directed trees to represent the partition of $\mathcal{U}$, and uses the roots of the trees as the designated representatives for each set in the partition. Two algorithmic techniques are used to speed up computations, namely UNION-BY-RANK and PATH-COMPRESSION (see, e.g. [5], chapter 22). The space overhead of the data structure is only $O(n)$ – each node u only stores a parent pointer $p(u)$ (which points to u if u is the root) and the rank of u. It is well-known that, starting from all sets in the partition being singletons, m UNION operations take amortized time $O((n + m)\alpha(m, n))$, where $\alpha(m, n)$ is a slowly growing function related to the inverse of the Ackermann function[25]. A significantly simpler analysis can be used to show that the amortized cost of m UNION operations is $O((m + n) \log^* n)$, where

$\log^* n$ is the iterated logarithm. Both functions are extremely slow growing, and for any reasonable values of m, n one can assume that $\alpha(n, m) \leq 3$ and $\log^* n \leq 5$. In our application to sparsification the universe $\mathcal{U}$ will be the set of nodes of the input graph and the sets in the partition will be the connected components of random samples of G.

A simple distributed implementation is as follows. Let n denote the number of elements in the universe. Let k denote the number of machines and let $h : [n] \to [k]$ denote a hash function. A naive implementation simply stores, for each $u \in [n]$, the parent pointer $p(u)$ on machine $h(u)$, and forwards the FIND request to machine $h(p(u))$ over the network. Amortized analysis of the complexity of FIND immediately translates into a bound on the network communication of the protocol. In particular, we obtain an implementation with $O(n/k)$ space requirement per machine and $O((m + n) \log^* n)$ network communication on a sequence of m edge arrivals on n nodes. This simple implementation, however, overlooks the important issue of load balancing. In particular, even though total communication is quite low, it could happen that a single node receives almost all requests, as illustrated by the following simple example. Let $\{r, u_1, \ldots, u_{n-1}\}$ denote the nodes of a complete graph. Then if edges $(u_i, r), i = 1, \ldots, n-1$ arrive first, making r the root of the UNION-FIND data structure, every subsequent edge of the form (u_i, u_j) leads to a request being routed through r. Thus, r receives $\Omega(n^2)$ requests. We now describe a simple modification that overcomes this issue and prove theoretical guarantees on the performance.

2.1 Load balancing

We show that the simple change to the UNION-FIND data structure known as immediate parent check or the zig-zag heuristic [22, 18] ensures that no vertex receives more than $2n$ requests. In particular, when an edge $e = (u, v)$ arrives the algorithm first checks if the immediate parent pointers of u and v are equal, and only forwards a FIND request to the parent of u and v respectively if the immediate parents are different:

Algorithm 1: LAZY-UNION(u, v)

```
1: if p(u) ≠ p(v) then
2:     r_u ← FIND (u)
3:     r_v ← FIND (v)
4:     if r_u ≠ r_v then
5:         UNION (u, v)
6:     end if
7: end if
```

We first show

Claim 1 *No* `<key, value>` *pair receives more than* $2n$ *parent queries.*

Proof: First note that once a vertex becomes a non-root vertex, it stays a non-root vertex. Consider a vertex w. We will show that w gets at most n parent request when it is the root vertex, and at most n requests after it becomes a non-root vertex.

By definition of LAZY-UNION a parent request originating from a pair (u, v) goes to a *root* vertex w only when at least one of u, v is a descendant of w and not its direct child in the UNION-FIND tree. After each call to LAZY-UNION both u and v become direct descendants of the root, so as long as w stays the root vertex, it cannot receive more than n parent queries.

Now suppose that w is not the root vertex. Every time w gets a parent request from its descendant u, u's parent pointer is rerouted so that u is no longer a descendant of w. Hence, w can receive at most n requests while it is a non-root vertex, which completes the proof. ■

We note that it is crucial for Claim 1 that FIND is only called from a UNION. This will be important later in section 4 when we give a distributed implementation of a one-pass sparsification scheme. We now prove

Lemma 2 *For any $\delta > 0$ no reducer receives more than $4n\lceil \frac{m \log^* n}{nk} \rceil \log(k/\delta) \approx 4(m/k) \log^* n \log(k/\delta)$ queries with probability at least $1 - \delta$.*

Proof: Consider an execution of UNION-FIND and denote for each vertex $u \in V$ let t_u denote the number of requests that u receives during the execution of the algorithm. By Claim 1 we have $t_u \leq 2n$ for all $u \in V$. Furthermore, by Lemma 3 we have $\sum_{u \in V} t_u = O(m \log^* n)$. Fix a machine i and let $I(u, i)$ denote the indicator variable that equals 1 if u is assigned to machine i and 0 otherwise. Then load of machine i is thus $\sum_{u \in V} t_u I(u, i)$, and we have for any $i = 1, \ldots, k$ by Chernoff bounds that

$$\textbf{Pr}\left[\text{load of machine } i > 4n \left\lceil \frac{m \log^* n}{nk} \right\rceil \log(k/\delta)\right] = \textbf{Pr}\left[\sum_{u \in V} t_u I(u, i) > 4n \left\lceil \frac{m \log^* n}{nk} \right\rceil \log(k/\delta)\right] < \delta/k.$$

Thus, no machine has a larger load with probability at least $1 - \delta$ by the union bound. ■

2.2 Race conditions

We now describe our implementation of UNION-FIND that ensures the absence of race conditions. This will be crucial to ensuring that our one-pass sparsification routine produces a sparsifier of small size. We first assign uniformly random id's from the range $[1 : n^3]$ to machines, so that the probability that two machines have the same id's is negligible. Once an edge (u, v) arrives, its processing is assigned to one of $h(u)$ and $h(v)$ that has the smallest id. Assuming wlog that $id(h(u)) < id(h(v))$, machine $h(u)$ stores the edge in a queue and emits FIND-ROOT requests from $h(u)$ and $h(v)$, which at some point come back with purported roots r_u and r_v of u's and v's components respectively. Note that these answers need not in general be accurate when $h(u)$ receives them. Machine $h(u)$ then initiates a *blocking call* to $h(r_v)$ and merges the roots if they are still roots of their components, and returns **connected**. Otherwise new FIND-ROOT requests are issued. The ability of UNION to return a value depending on whether or not the call resulted in two distinct components merging will be important for the single pass sparsification routine, but not for multi-pass sparsification. In particular, we stress that the multi-pass implementation can be made *completely asynchronous* by avoiding blocking calls.

First, we note that the scheme above does not produce deadlocks since a node can only wait for a node with higher id[1]. Also,

Lemma 3 *The distributed parallel scheme produces a correct UNION-FIND data structure at the end of the process. The amortized cost is $O(\log^* n)$ per FIND.*

Proof: Correctness is clear. Efficiency follows by repeating the charging argument in the classical proof of $O(\log^* n)$ amortized complexity of UNION-FIND . ■

3 Sparsification

In this section we demonstrate an application of our distributed implementation of UNION-FIND to graph sparsification via a *refinement sampling scheme*. We will give a *one pass distributed implementation* of sparsification using the UNION-FIND data structure developed earlier. In fact, by exploiting the properties of our refinement sampling scheme, we will be able to preserve the favorable load balancing properties demonstrated in section 2.

We will denote by $G(V, E)$ the input undirected graph with vertex set V and edge set E with $|V| = n$ and $|E| = m$. For any $\epsilon > 0$, we say that a weighted graph $G'(V, E')$ is an *ϵ-sparsification* of G if every

[1]This only holds when there are no duplicate ids, i.e. with high probability. However, a worst-case guarantee can be obtained by simply resampling the id of the vertex that initiates a blocking call if its ids is the same as its peer's id.

(weighted) cut in G' is within $(1 \pm \epsilon)$ of the corresponding cut in G. Given any two collections of sets that partition V, say S_1 and S_2, we say that S_2 is a *refinement* of S_1 if for any $X \in S_1$ and $Y \in S_2$, either $X \cap Y = \emptyset$ or $Y \subset X$. In other words, $S_1 \cup S_2$ form a laminar set system.

3.1 Benczúr-Karger Sampling Scheme

We say that a graph is k-*connected* if the value of each cut in G is at least k. The Benczúr-Karger sampling scheme uses a more strict notion of connectivity, referred to as *strong connectivity*, defined as follows:

Definition 4 *[4] A* k-strong component *is a maximal* k*-connected vertex-induced subgraph. The* strong connectivity *of an edge* e*, denoted by* s_e*, is the largest* k *such that a* k*-strong component contains* e*.*

Note that the set of k-strong components form a partition of the vertex set of G, and the set of $k+1$-strong components forms a refinement this partition. We say e is k-*strong* if its strong connectivity is k or more, and k-*weak* otherwise. The following simple lemma will be useful in our analysis.

Lemma 5 *[4] The number of* k*-weak edges in a graph on* n *vertices is bounded by* $k(n-1)$*.*

The sampling algorithm relies on the following result:

Theorem 6 *[4] Let* G' *be obtained by sampling edges of* G *with probability* $p_e = \min\{\frac{\rho}{\epsilon^2 s_e}, 1\}$*, where* $\rho = 16(d+2)\ln n$*, and giving each sampled edge weight* $1/p_e$*. Then* G' *is an* ϵ*-sparsification of* G *with probability at least* $1 - n^{-d}$*. Moreover, expected number of edges in* G' *is* $O(n \log n)$*.*

It follows easily from the proof of theorem 6 in [4] that if we sample using an *underestimate* of edge strengths, the resulting graph is still an ϵ-sparsification.

Corollary 7 *Let* G' *be obtained by sampling each edge of* G *with probability* $\tilde{p}_e \geq p_e$ *and and give every sampled edge* e *weight* $1/\tilde{p}_e$*. Then* G' *is an* ϵ*-sparsification of* G *with probability at least* $1 - n^{-d}$*.*

In what follows we will consider unweighted graphs to simplify notation. The results obtained can be easily extended to the polynomially weighted case as outlined in Remark 15 in Appendix A.

4 Refinement Sampling

We start by introducing the idea of refinement sampling that gives a simple algorithm for efficiently computing a BK-sample, and serves as a building block for our streaming algorithms.

To motivate refinement sampling, let us consider the simpler problem of identifying all edges of strength at least k in the input graph $G(V, E)$. A natural idea to do so is as follows: (a) generate a graph G' by sampling edges of G with probability $\tilde{O}(1/k)$, (b) find connected components of G', and (c) output all edges $(u, v) \in E$ as such that u and v are in the same connected component in G'. The sampling rate of $\tilde{O}(1/k)$ suggests that if an edge (u, v) has strong connectivity below k, the vertices u and v would end up in different components in G', and conversely, if the strong connectivity of (u, v) is above k, they are likely to stay connected and hence output in step (c). While this process indeed filters out most k-weak edges, it is easy to construct examples where the output will contain many edges of strength 1 even though k is polynomially large (a star graph, for instance). The idea of refinement sampling is to get around this by successively *refining* the sample obtained in the final step (c) above.

In designing our algorithm, we will repeatedly invoke the subroutine $\textsc{Refine}(S, p)$ that essentially implements the simple idea described above.

Function: REFINE(S, p)

Input: Partition S of V, sampling probability p.

Output: Partition S' of V, a refinement of S.

1. Take a uniform sample E' of edges of E with probability p.

2. For each $U \in S, U \subseteq V$ let $C(U)$ be the set of connected components of U induced by E'.

3. Return $S' := \cup_{U \in S} C(U)$.

We note that REFINE can be implemented using a single pass over the set of edges. A scheme of refinement relations between $S_{l,k}$ is given in Fig. 3.

The **refinement sampling** algorithm computes partitions $S_{l,j}$ for $l = 1, \ldots, L$ and $j = 0, 1, \ldots, K$. Here $L = \log(2n)$ is the number of strength levels (the factor of 2 is chosen for convenience to ensure that $S_{L,K}$ consists of isolated vertices whp), K is a parameter which we call the *strengthening* parameter. Also, we choose a parameter $\phi > 0$, which we will refer to as the oversampling parameter. For a partition S, let $X(S)$ denote all the edges in E which have endpoints in two different sets in S. The partitions are computed as follows:

Algorithm 1 (Refinement Sampling)

Initialization: $S_{l,0} = \{V\}$ for $l = 1, \ldots, L$.

1. Set $k := 1$

2. For each l, $1 \leq l \leq L$, set $S_{l,k} :=$ REFINE$(S_{l,k-1}, 2^{-l})$.

3. Set $k := k + 1$. If $k < K$, go to step 1.

4. For each $e \in E$ define $L(e) = \min\{l : e \in X(S_{l,K})\}$. Sample edge e with probability $z(e) = \min\{1, \frac{\phi}{\epsilon^2 2^{L(e)}}\}$ and assign it weight $1/z(e)$. Let $R(\phi, K)$ denote the set of edges sampled during this step; we call this the refinement sample of G.

We note here that since the refinement sampling scheme only uses the UNION-FIND data structure, it translates directly to a distributed implementation that inherits good load balancing properties proved in section 2, with race conditions handled as described in section 2. Recall that due to the asynchronous nature of UNION-FIND implementation, UNION may return **connected** if the nodes that it was called on were merged in the process. In order to implement the refinement sampling scheme given above, it is sufficient to ignore the output of UNION (in contrast to the one-pass scheme that we will give in section 5). In particular, the implementation can be made *completely asynchronous*.

The following two lemmas relate the probabilities $z(e)$ to the sampling probabilities used in the Benczúr-Karger sampling scheme.

Lemma 8 *For any $K > 0$, with probability at least $1 - Kn^{-d}$ every edge e satisfies $z(e) \leq 4\phi\rho/(\epsilon^2 s_e)$.*

Lemma 9 *If $K > \log_{4/3} n$, then $2^{-L(e)+1} \geq 1/(2s_e)$ for every $e \in E(G)$ with probability at least $1 - Ke^{-(n-1)/100}$.*

The proofs of Lemma 8 and Lemma 9 are given in Appendix A.

Theorem 10 *Let G' be the graph obtained by running Algorithm 1 with $\phi := 4\rho$. Then G' has $O(n \log^2 n/\epsilon^2)$ edges in expectation, and is an ϵ-sparsification of G with probability at least $1 - n^{-d+1}$.*

Proof: We have from lemma 9 and the choice of ϕ that the sampling probabilities dominate those used in Benczúr-Karger sampling with probability at least $1 - Ke^{-(n-1)/100}$. Hence, by corollary 7 we have that every cut in G' is within $1 \pm \epsilon$ of its value in G with probability at least $1 - Ke^{-(n-1)/100} - n^{-d}$. The expected size of the sample is $O(n \log^2 n/\epsilon^2)$ by lemma 8 together with the fact that $\rho = O(\log n)$. The probability of failure of the estimate in lemma 9 is at most Kn^{-d}, so all bounds hold with probability at least $1 - Kn^{-d} + Ke^{-(n-1)/100} - n^{-d} > 1 - n^{-d+1}$ for sufficiently large n. The high probability bound on the number of edges follows by an application of the Chernoff bound. ■

It follows from the discussion above that Algorithm 1 produces a sparsifier in can be constructed in $O(\log n)$ passes of REFINE using $O(\log n)$ space per node and $O(\log^2 n \log^* n)$ work per edge.

5 A One-pass $\tilde{O}(n+m)$-Time Algorithm for Graph Sparsification

In this section we convert Algorithm 1 obtained in the previous section to a one-pass algorithm. We will design a one-pass algorithm that produces an ϵ-sparsifier with $O(n \log^3 n/\epsilon^2)$ edges. The main difficulty is that in going from $O(\log n)$ passes to a one-pass algorithm, we need to introduce and analyze new dependencies in the sampling process.

As before, the algorithm maintains connectivity data structures $D_{l,k}$, where $1 \leq l \leq L$ and $1 \leq k \leq K$. In addition to indexing $D_{l,k}$ by pairs (l,k) we shall also write D_J for $D_{l,k}$, where $J = K(l-1)+k$, so that $1 \leq J \leq LK$. This induces a natural ordering on $D_{l,k}$, illustrated in Fig. 4, that corresponds to the structure of refinement relations. We will assume for simplicity of presentation that $D_0 = D_{1,0}$ is a connectivity data structure in which all vertices are connected. For each edge e, $1 \leq \ell \leq L$, and $1 \leq k \leq K$, we define an independent Bernoulli random variable $A'_{l,k,e}$ with $\mathbf{Pr}[A'_{l,k,e} = 1] = 2^{-l}$. The algorithm is as follows:

Algorithm 2 (A One-Pass Sparsifier)

Input: Edges of G streamed in adversarial order: $(e_1, \ldots, e_m)$.

Output: A sparsification G' of G.

Initialization: Set $E' := \emptyset$.

1. Set $t = 1$.

2. For all $J = 1, \ldots, LK$ ($J = (l,k)$)

3. Add $e_t = (u_t, v_t)$ to D_J if $A'_{l,k,e} = 1$ and u_t and v_t are connected in D_{J-1}.

4. Define $L'(e_t)$ as the minimum l such that u_t and v_t are not connected in $D_{l,K}$. Set $z'(e_t) := \min\left\{1, \frac{4\rho}{\epsilon^2 2^{L'(e_t)}}\right\}$. Output e_t with probability $z'(e_t)$, giving it weight $1/z'(e_t)$.

5. Set $t := t+1$. Go to step 2 if $t \leq m$.

Note that Algorithm 2 admits a simple distributed implementation using UNION-FIND queries described in section 2. An incoming edge is simply forwarded from one data structure to the next while the endpoints are in the same component or a coin flip comes up heads. It is important here that the edge is only forwarded to the next data structure if the corresponding coin comes up heads, so that if the roots end up being different, a UNION request is emitted and the load balancing result of Claim 1 still applies. It should be noted here that a distributed implementation crucially uses the fact that a call to UNION may return **connected**. If that happens, the edge is forwarded to the next UNION-FIND data structure.

Informally, Algorithm 2 underestimates strength of some edges until the data structures $D_{l,k}$ become properly connected but proceeds similarly to Algorithms 1 and 2 after that. Our main goal in the rest of the

section is to show that this underestimation of strengths does not lead to a large increase in the size of the sample.

Note that not all $LK = \Theta(\log^2 n)$ coin tosses $A'_{l,k,e}$ per edge are necessary for an implementation of Algorithm 2. However, the random variables $A'_{l,k,e}$ are useful for analysis purposes. We now show that Algorithm 2 outputs a sparsification G' of G with $O(n \log^3 n/\epsilon^2)$ edges whp.

Lemma 11 *For any $\epsilon > 0$, w.h.p. the graph G' is an ϵ-sparsification of G.*

Proof: We can couple behaviors of Algorithms 1 and 3 using the coin tosses $A'_{l,k,e}$ to show that $L(e) \geq L'(e)$ for every edge e, i.e. $z'(e) \geq z(e)$. Hence G' is a sparsification of G by Corollary 7. ■

It remains to upper bound the size of the sample. The following lemma is crucial to our analysis; its proof is deferred to the Appendix B due to space limitations.

Lemma 12 *Let $G(V, E)$ be an undirected graph. Consider the execution of Algorithm 2, and for $1 \leq J \leq LK$ where $J = (l, k)$, let X^J denote the set of edges $e = (u, v)$ such that u and v are connected in D_{J-1} when e arrives. Then $|E \setminus X^J| = O(K2^l n)$ with high probability.*

Lemma 13 *The number of edges in G' is $O(n \log^3 n/\epsilon^2)$ with high probability.*

Proof: Recall that Algorithm 2 samples an edge $e_t = (u_t, v_t)$ with probability $z'(e_t) = \min\left\{1, \frac{4\rho}{\epsilon^2 2^{L'(e_t)}}\right\}$, where $L'(e_t)$ is the minimum l such that u_t and v_t are not connected in $D_{l,K}$. As before, for $J = (l, k)$, we denote by X^J the set of edges $e = (u, v)$ such that u and v are connected in D_{J-1} when e arrives. Note that w.h.p. $X^{(L,1)} = \emptyset$ w.h.p. by our choice of $L = \log(2n)$. For each $1 \leq l \leq L$, let $Y_l = X^{(l,1)} \setminus X^{(l+1,1)}$. We have by Lemma 12 that $\sum_{1 \leq j \leq l} |Y_j| = O(K2^l n)$ w.h.p. Also note that edges in Y_l are sampled with probability at most $\frac{4\rho}{\epsilon^2 2^{l-1}}$. Hence, we get that the expected number of edges in the sample is at most

$$\sum_{l=1}^{L} |Y_l| \cdot \frac{4\rho}{\epsilon^2 2^{l-1}} = O\left(\sum_{l=1}^{L} K2^l n \cdot \frac{4\rho}{\epsilon^2 2^{l-1}}\right) = O(n \log^3 n/\epsilon^2).$$

The high probability bound now follows by standard concentration inequalities. ■

Finally, we have the following theorem.

Theorem 14 *For any $\epsilon > 0$ and $d > 0$, there exists a one-pass algorithm that given the edges of an undirected graph G streamed in adversarial order, produces an ϵ-sparsifier G' with $O(n \log^3 n/\epsilon^2)$ edges with high probability. The algorithm takes $O(\log^2 n \log^* n)$ work per edge and uses $O(\log^2 n)$ space per node.*

Proof: Lemma 11 and Lemma 13 together establish that G' is an ϵ-sparsifier G' with $O(n \log^3 n/\epsilon^2)$ edges, and the runtime bounds follow immediately. ■

6 Experimental evaluation

In this section we present results of empirical evaluation of our algorithm on synthetic and real-world data.

6.1 Load balancing

The first set of experiments supports the load-balancing guarantees proven in section 2. In this experiment we simulated the one-pass sparsification algorithm given in section 5. The input graph that was used consists of all reciprocated edges among 5 Million nodes sampled from the Twitter network; the sampling was non-uniform to preempt any inference regarding the average density of the Twitter social graph.

The graph has $\approx 5M$ nodes and $\approx 100M$ edges. The nodes of the graph were distributed among 100 machines using a hash function. Then a 10 level refinement sampling scheme was executed with the number of reinforcement rounds set to $K = 2$. The strength parameter was varied geometrically as 0.7^j, where $j = 1, \ldots, L$ is the strength level.

We measured the average and maximum number of accesses to (a) `<key, value>` pairs and (b) individual machines. The maximum number of accesses to a `<key, value>` pair was 127416, which is significantly smaller than the upper bound of $2n \approx 10M$ proved in Claim 1. All $K \cdot L$ UNION-FIND data structures used the same hash functions to map vertices to machines. We measured the load on a machine coming from (a) following pointers in forwarded FIND requests and (b) edges incident on nodes that map to the machine, which we refer to as *local* requests. Thus, while (a) is a related to the overhead of UNION-FIND , (b) is a function of the variance of the degree sequence of the graph. The results are given in Fig. 1, where $K \cdot L = 20$ UNION-FIND data structures are ordered according to the natural ordering shown in Fig. 4. Note that the maximum for forwarded FIND requests is never more than twice the average, and the maximum load for local operations does not exceed the average load by more than 25%. Also, the load from forwarded FIND requests is at least a factor of 10 smaller than the load from local requests, so the second plot dominates the picture.

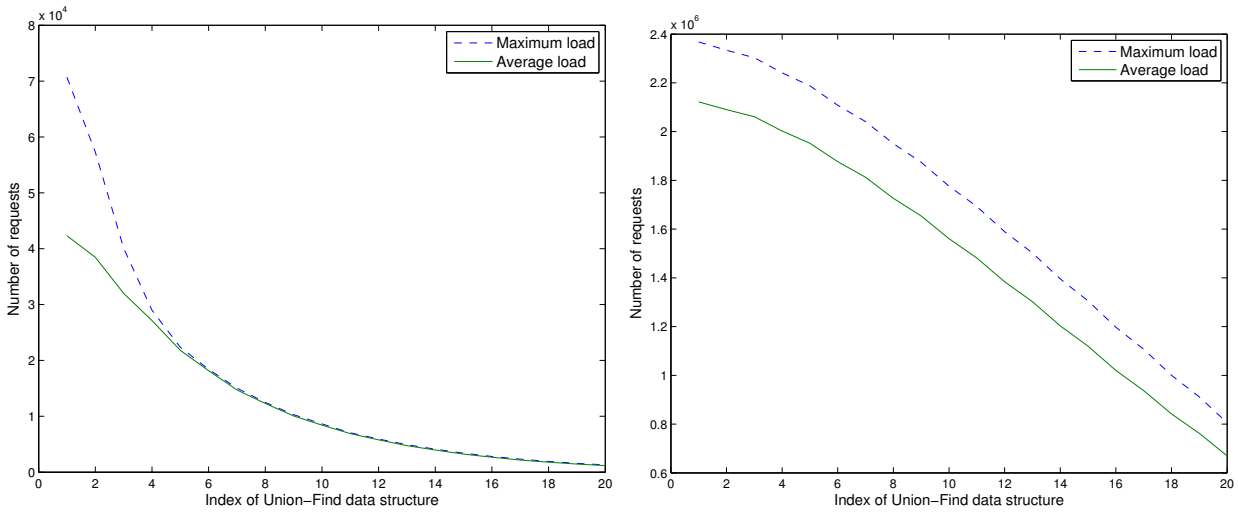

Figure 1: Load on machines caused by forwarded FIND requests (left) and local requests (right)

6.2 Precision of refinement sampling scheme

In order to validate the approximation guaranteed of the refinement sampling scheme, we ran experiments on a synthetic graph with a nontrivial cut structure and compared the structure of strongly connected components recovered by the algorithm to the true one. The experiments were carried out with the single pass and multipass refinement sampling schemes respectively.

We generated a random graph on 800000 nodes using a three-level partition of the vertex set. The partition was given by a three-level tree with branching factor $20, 200, 200$ on the first, second and third levels respectively (and $20 \cdot 200 \cdot 200$ leaves at level 4). The edges were generated as follows. For each level $l = 1, 2, 3$, where $l = 1$ is the root, for each u at level l let the children of u be $c_1, \ldots, c_k$ (note that $k = 20$ if $l = 1$ and $k = 200$ otherwise). For each u the graph obtained by contracting subtrees T_{c_i} to supernodes was an Erdös-Renyi graph with expected degree d_l. The endpoints of edges incident on supernodes T_{c_i} were uniformly distributed over the leaves of T_{c_j}. We used $d_1 = 6, d_2 = 40$ and $d_3 = 160$. Thus, the density increases from the root to the leaves, and the strongly connected components are simply the subtrees at level

l for $l = 1, 2, 3$.

For the multipass experiment the refinement sampling scheme was used with $K = 2, L = 7$ and sampling probabilities given in Fig. 2 (left panel). For each sampling probability p and for each $j = 1, \ldots, n$ let $c(j, p)$ denote the number of vertices belonging to component of size at most j in the UNION-FIND data structure corresponding to the final reinforcement step for probability p. Note that in the synthetic graph that we use ideally for each p one has that $c(j, p)$ is a step function in j, with steps at $200, 200 \cdot 200$ and $20 \cdot 200 \cdot 200$, corresponding to the sizes of subtrees.

The plot of the functions $c(j, p)$ size is given in Fig. 2 (left panel). The x-axis is log-scale in order to ensure that $\log_{200} 200 = 1$ and $\log_{200} 200^2 = 2$ are the reference points for the sizes of subtrees, i.e. the sizes of true strongly connected components. We note here that in the multi-pass scheme, a larger number of strength levels only improves the precision of the scheme (at the expense of increasing the number of passes). This, however, is not the case for the one-pass scheme, where choosing a finer grid of strength levels is not necessarily optimal (in fact, geometrically decreasing strengths are a good choice, as follows from the analysis of Algorithm 2). This explains our choice of a finer grid of strength levels for the multi-pass experiment and geometrically decreasing strengths for the single-pass experiment.

We note that the refinement sampling scheme provides a factor 3 approximation to strong connectivities of $\geq 95\%$ of the edges of the graph. Indeed, note that the number of nodes in components of size at least 200^2 drops from $\geq 95\%$ to 0 between sampling probability 0.1 and 0.05, whereas the expected degree is 40, so that sampling at rate $\lambda/40 = 0.025\lambda, \lambda \geq 1$ is expected to keep $1 - e^{-\Theta(\lambda)}$ vertices in a single connected component. Similarly, the number of vertices that belong to connected components of size 200 drops from $\geq 95\%$ to 0 in the narrow range 0.025 to 0.01, consistent with expected degree 160.

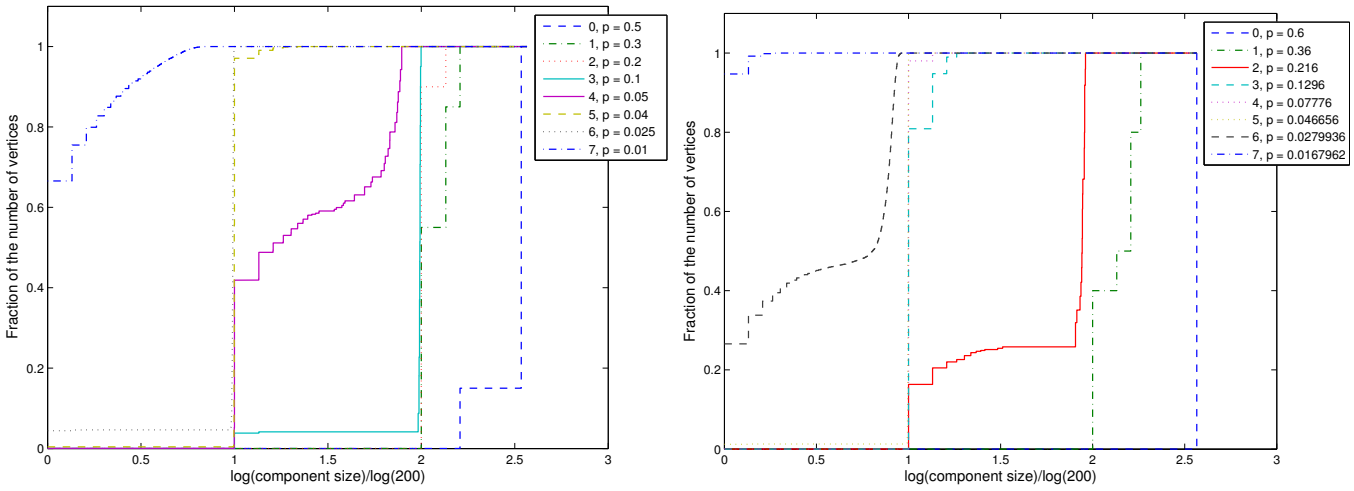

Figure 2: Distribution of vertices among strength levels in the multipass (left) and single pass scheme (right)

For the second experiment we used the single pass sparsification scheme on the same graph. The parameters of the single-pass sparsification were $K = 1, L = 6$ with sampling rate 0.6, so that strength levels are given by $0.6^j, j = 1, \ldots, L$ (see Fig. 2, right panel). As expected, the precision of this scheme is lower than that of the multipass scheme. The second level components become disconnected at sampling rate close to 0.216, and the third level components become disconnected at sampling rate ≈ 0.03, consistent with expected degrees 40 and 160 with slightly larger constant factors.

References

[1] http://hadoop.apache.org.

[2] http://tech.backtype.com/preview-of-storm-the-hadoop-of-realtime-proce.

[3] K. Ahn and S. Guha. On graph problems in a semi-streaming model. *ICALP*, 2009.

[4] András A. Benczúr and David R. Karger. Approximating *s-t* minimum cuts in $\tilde{O}(n^2)$ time. *STOC*, 1996.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[7] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348:207–216, 2005.

[8] W. Fung, R. Hariharan, N. Harvey, and D. Panigrahi. A general framework for graph sparsification. *STOC*, 2011.

[9] A. Goel, M. Kapralov, and S. Khanna. Graph sparsification via refinement sampling. `http://arxiv.org/abs/1004.4915`, 2010.

[10] Navin Goyal, Luis Rademacher, and Santosh Vempala. Expanders via random spanning trees, 2009.

[11] M. Kapralov and R. Panigrahy. Spectral sparsification via random spanners. *to appear ITCS*, 2012.

[12] D. Karger. Random sampling in cut, flow, and network design problems. *STOC*, 1994.

[13] D. Karger and M. Levine. Random sampling in residual graphs. *STOC*, 2002.

[14] Jonathan A. Kelner and Alex Levin. Spectral sparsification in the semi-streaming setting. *STACS*, 2011.

[15] R. Khandekar, S. Rao, and V. Vazirani. Graph partitioning using single commodity flows. *STOC*, 2006.

[16] I. Koutis, G. Miller, and R. Peng. Approaching optimality for solving sdd linear systems. *FOCS*, 2010.

[17] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. *SIGMOD '10*.

[18] Fredrik Manne and Md. Mostofa Ali Patwary. A scalable parallel union-find algorithm for distributed memory computers. *PPAM*, 77, 2009.

[19] S. Muthukrishnan. *Data streams: algorithms and applications*. Now publishers, 2006.

[20] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[21] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4:distributed stream computing platform. *Proceedings of the International Conference on Data Mining Workshops*, pages 170–177, 2010.

[22] V. Osipov, P. Sanders, and J. Singler. The filter-kruskal minimum spanning tree algorithm. *ALENEX*, pages 52–61, 2009.

[23] D.A. Spielman and N. Srivastava. Graph sparsification by effective resistances. *STOC*, 2008.

[24] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. *STOC '04*, 2004.

[25] R. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 1975.

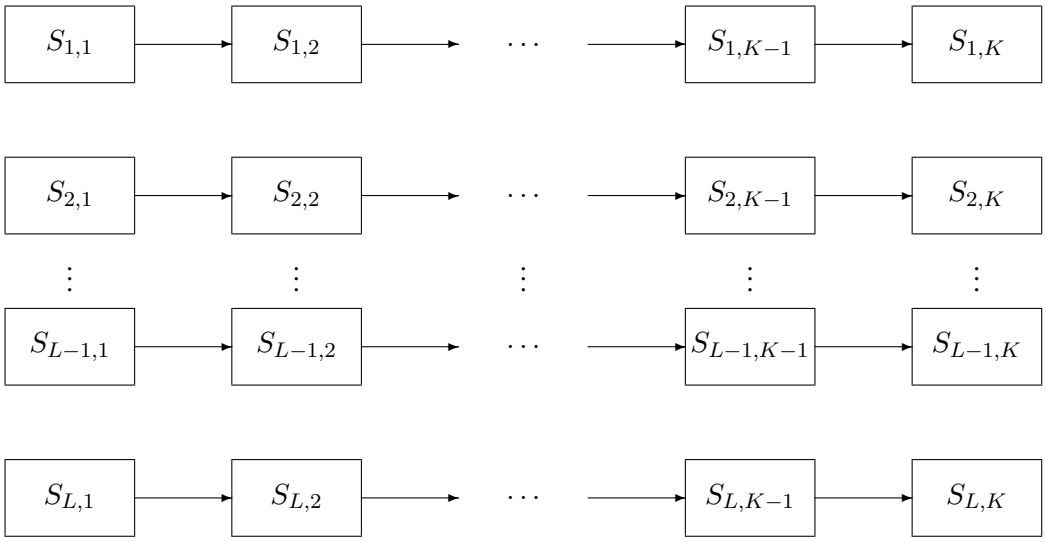

Figure 3: Scheme of refinement relations between partitions for Algorithm 1.

A Proof from section 4

Proof of Lemma 8: Consider an edge e with strong connectivity s_e, and let C denote the s_e-strongly connected component containing e. By Theorem 6, sampling with probability $\min\{4\rho/s_e, 1\}$ preserves all cuts up to $1\pm\frac{1}{2}$ in C with probability at least $1-n^{-d}$. Hence, all s_e-strongly connected components stay connected after K passes of REFINE for all $l > 0$ such that $2^{-l} \geq 4\rho/s_e$, yielding the lemma. ■

Proof of Lemma 9: Consider a level l such that $p = 2^{-l} < 1/(2s_e)$. Let H be the graph obtained by contracting all (s_e+1)-strong components in G into supernodes. Since H contains only (s_e+1)-weak edges, the number of edges is at most $s_e(n-1)$ by Lemma 5. As the expected number of (s_e+1)-weak edges in the sample is at most $(n-1)/2$, by Chernoff bounds, the probability that the number of (s_e+1)-weak edges in the sample exceeds $3(n-1)/4$ is at most $(e^{1/4}(5/4)^{-5/4})^{-(n-1)/2} < e^{-(n-1)/100}$. Thus at least one quarter of the supernodes get isolated in each iteration. Hence, no (s_e+1)-weak edge survives after $K = \log_{4/3} n$ rounds of refinement sampling with probability at least $1 - Ke^{-(n-1)/100}$. Since $L(e)$ was defined as the least l such that $e \in X(S_{l,K})$, the endpoints of e were connected in $S_{L(e)-1,K}$, so $2^{-L(e)+1} \geq 1/(2s_e)$. ■

Remark 15 *Algorithms 1-2 can be easily extended to graphs with polynomially bounded integer weights on edges. If we denote by W the largest edge weight, then it is sufficient to set the number of levels L to $\log(2nW)$ instead of $\log(2n)$ and the number of passes to $\log_{4/3} nW$ instead of $\log_{4/3} n$. A weighted edge is then viewed as several parallel edges, and sampling can be performed efficiently for such edges by sampling directly from the corresponding binomial distribution.*

B Proof of Lemma 12

We denote the edges of G in their order in the stream by $E = (e_1, \ldots, e_m)$. In what follows we shall treat edge sets as ordered sets, and for any $E_1 \subseteq E$ write $E \setminus E_1$ to denote the result of removing edges of E_1 from

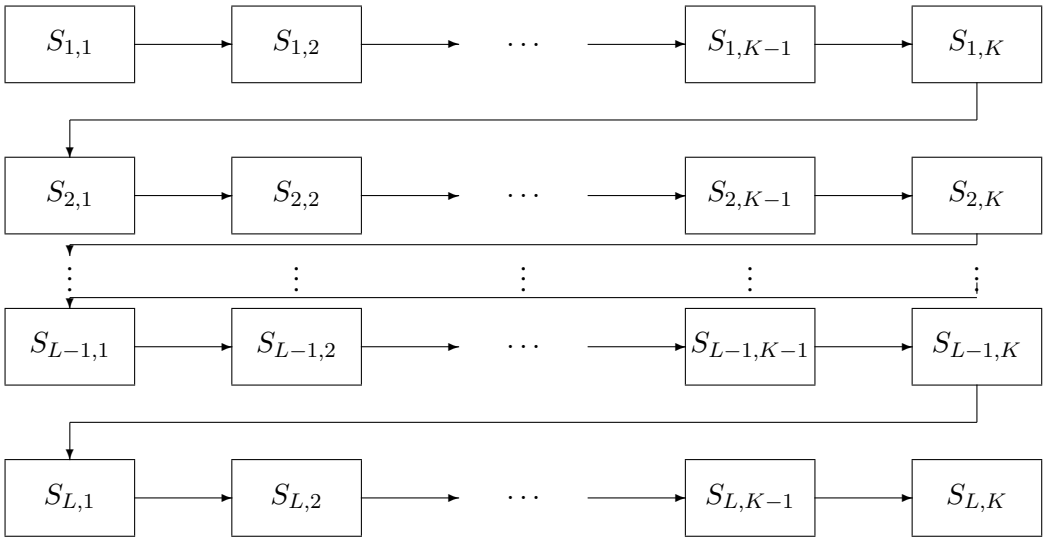

Figure 4: Scheme of refinement for Algorithm 2.

E while preserving the order of the remaining edges. For a stream of edges E we shall write E_t to denote the set of the first t edges in the stream.

For a κ-connected component C of a graph G we will write $|C|$ to denote the number of vertices in C. Also, we will denote the result of sampling the edges of C uniformly at random with probability p by C'. The following simple lemma will be useful in our analysis:

Lemma 16 *Let C be a κ-connected component of G for some positive integer κ. Denote the graph obtained by sampling edges of C with probability $p \geq \lambda/\kappa$ by C'. Then the number of connected components in C' is at most $\gamma|C|$ with probability at least $1 - e^{-\eta|C|}$, where $\gamma = (7/8 + e^{-\lambda/2}/8)$ and $\eta = 1 - e^{-\lambda/2}$.*

Proof: Choose $A, B \subset V(C)$ so that $A \cup B = V(C), A \cap B = \emptyset$, $|A| \geq |V(C)|/2$ and for every $v \in A$ at least half of its edges that go to vertices in C go to B. Note that such a partition always exists: starting from any arbitrary partition of vertices of C, we can repeatedly move a vertex from one side to the other if it increases the number of edges going across the partition, and upon termination, the larger side corresponds to the set A. Denote by Y the number of vertices of A that belong to components of size at least 2. Note that Y can be expressed as sum of $|A|$ independent $0/1$ Bernoulli random variables. Let $\mu := \mathbf{E}[Y]$; we have that $\mu \geq |A|(1 - (1 - \lambda/\kappa)^{\kappa/2}) \geq |A|(1 - e^{-\lambda/2})$. We get by the Chernoff bound that $\mathbf{Pr}[Y \leq |A|(1 - e^{-\lambda/2})/2] \leq e^{-2\mu} \leq e^{-|C|(1-e^{-\lambda/2})} = e^{-\eta|C|}$. Hence, at least a $(1 - e^{-\lambda/2})/4$ fraction of the vertices of C are in components of size at least 2. Hence, the number of connected components is at most a $1 - (1 - e^{-\lambda/2})/8 = 7/8 + e^{-\lambda/2}/8 = \gamma$ fraction of the number of vertices of C. ■

Proof of Lemma 12: The proof is by induction on J. We prove that w.h.p. for every $J = (l, k)$ one has $|E \setminus X^J| \leq \sum_{1 \leq J' = (l',k') \leq J-1} c_1 2^{l'} n$ for a constant $c_1 > 0$.

Base: $J = 1$ Since everything is connected in D_0 by definition, the claim holds.

Inductive step: $J \to J+1$ The outline of the proof is as follows. For every $J = (l,k)$ we consider the edges of the stream that the algorithm tries to add to D_J, identify a sequence of 2^l-strongly connected components $C_0, C_1 \ldots$ in the partially received graph, and use lemma 16 to show that the number of connected components decreases fast because only a small fraction of vertices in the sampled 2^l-strongly connected components are isolated. We thus show that, informally, it will take $O(2^l n)$ edges to make the connectivity data structure D_J in Algorithm 2 connected. The connected components C_s are defined by induction on s. The vertices of C_s are elements of a partition P_s of the vertex set V of the graph G. We shall use an auxiliary sequence of graphs which we denote by H_s^t.

Let P_0 be the partition consisting of isolated vertices of V. We treat the base case $s = 0$ separately to simplify exposition. We use the definition of γ and η from lemma 16 with $\lambda = 1$ since we are considering 2^l-connected components when $J = (k,l)$.

Base case: $s = 0$**.** Set $H_0^t = (P_0, \{e_1, \ldots, e_t\})$, i.e. H_0^t is the partially received graph up to time t. Let t_0^* be the the first value of t such that $s_{H_0^t}(e_t) \geq 2^l$. This means that $e_{t_0^*}$ belongs to a 2^l-strongly connected component in $H_0^{t_0^*}$. Note that this component does not contain any (2^l+1)-strongly connected components. Denote this component by C_0 (note that the number of edges in C_0 is at most $2^l|C_0|$ by lemma 5). Denote the random variables that correspond to sampling edges of C_0 by R_0. Let X_0 be an indicator variable that equals 1 if the number of connected components in C_0' is at most $\gamma|C_0|$ and 0 otherwise. By lemma 16 we have that $\mathbf{Pr}[X_0 = 1] \geq 1 - e^{-\eta|C_0|}$.

For a partition P denote $\text{diag}(P) = \{(u,u) : u \in P\}$. Define P_1 by merging partitions of P_0 that belong to connected components in C_0' if $X_0 = 1$ and as equal to P_0 otherwise. Let $E^1 = E \setminus (E(C_0) \cup \text{diag}(P_1))$, i.e. we remove edges of C_0 and also edges that connect vertices that belong to the same partition in P_1. Note that we can safely remove these edges since their endpoints are connected in D_J when they arrive. Define $H_1^t = (P_1, E_t^1)$, i.e. H_1^t is the partially received graph on the modified stream of edges.

Inductive step: $s \to s+1$**.** As in the base case, let t_s^* be the the first value of t such that $s_{H_s^t}(e_t) \geq 2^l$. This means that $e_{t_s^*}$ belongs to a 2^l-connected component in $H_s^{t_s^*}$. Denote this component by C_s(note that the number of edges in C_s is at most $2^l|C_s|$ by lemma 5). Denote the random variables that correspond to sampling edges of C_s by R_s. Let X_s be an indicator variable that equals 1 if the number of connected components in C_s' is at most $\gamma|C_s|$ and 0 otherwise. By lemma 16 we have that $\mathbf{Pr}[X_s = 1] \geq 1 - e^{-\eta|C_s|}$. Define P_{s+1} by merging together vertices that belong to connected components in C_s'. Let $E^{s+1} = E^s \setminus (E(C_s) \cup \text{diag}(P_s))$. Denote $H_s^t = (P_s, E_t^s)$.

It is important to note that at each step s we only flip coins R_s that correspond to edges in $E(C_s)$, and delete only those edges from E^s. While there may be edges going across partitions P_s for which we do not perform a coin flip, there number is bounded by $O(2^l n)$ since these edges do not contain a 2^l-connected component.

Note that for any $s > 0$ the number of connected components in P_s is at most

$$n - \sum_{j=1}^{s} (1-\gamma)|C_j|X_j.$$

We now show that it is very unlikely that $\sum_{j=1}^{s} |C_j|X_j$ is more than a constant factor smaller than $\sum_{j=1}^{s} |C_j|$, thus showing that the number of connected components cannot be more than 1 when $\sum_{j=1}^{s} |C_j| \geq \frac{cn}{1-\gamma}$ for an appropriate constant $c > 0$.

For any constant $d > 0$ define $I^+ = \{i \geq 0 : |C_i| > ((d+2)/\eta)\log n\}$ and $I^- = \{i \geq 0 : |C_i| \leq ((d+2)/\eta)\log n\}$. Also define $Z_i^+ = \sum_{0 \leq j \leq i, j \in I^+} X_j|C_j|$, $Z_i^- = \sum_{0 \leq j \leq i, j \in I^-} X_j|C_j| - |C_j|(1-e^{-\eta|C_j|})$.

First note that one has $\mathbf{Pr}[X_j = 1] \geq 1 - n^{-d-2}$ for any $j \in I^+$ by lemma 16. Hence, it follows by taking the union bound that $i \leq n^2$ one has $\mathbf{Pr}[Z_i^+ = \sum_{j \in I^+, j \leq i} |C_j|] \geq 1 - n^{-d}$.

We now consider Z_i^-. Note that Z_i^-'s define a martingale sequence with respect to $R_{i-1}, \ldots, R_0$: $\mathbf{E}[Z_i^- | R_{i-1}, \ldots, R_0] = Z_{i-1}^-$. Also, $|Z_i^- - Z_{i-1}^-| \leq ((d+2)/\eta) \log n$ for all i. Hence, by Azuma's inequality one has

$$\mathbf{Pr}[Z_i^- < t] < \exp\left(-\frac{t^2}{2i(((d+2)/\eta)\log n)^2}\right).$$

Now consider the smallest value τ such that $\sum_{j \leq \tau} |C_j| = \sum_{j \leq \tau, j \in I^+} |C_j| + \sum_{j \leq i, j \in I^-} |C_j| = S^+ + S^- \geq \frac{4n}{(1-e^{-2\eta})(1-\gamma)}$. Note that $\tau < n/(2(1-e^{-2\eta})(1-\gamma))$ since $|C_i| \geq 2$. If $S^+ \geq \frac{2n}{(1-e^{-2\eta})(1-\gamma)} \geq 2n/(1-\gamma)$, then we have that $Z_\tau^+ = S^+ > 2n/(1-\gamma)$ with probability at least $1 - n^{-d}$. Thus,

$$n - \sum_{j=1}^{\tau} (1-\gamma)|C_j| X_j \leq n - (1-\gamma) Z_\tau^+ \leq 0.$$

Otherwise $S^- \geq \frac{2n}{(1-e^{-2\eta})(1-\gamma)}$ and by Azuma's inequality we have

$$\mathbf{Pr}[Z_\tau^- < -n] < \exp\left(-\frac{n^2}{2\tau(((d+2)/\eta)\log n)^2}\right) \leq \exp\left(-\frac{n}{(((d+2)/\eta)\log n)^2}\right) < n^{-d}.$$

Since $|C_i| \geq 2$, we have $|C_i|(1 - e^{-\eta|C_i|}) \geq |C_i|(1 - e^{-2\eta})$ and thus we get

$$\begin{aligned} n - \sum_{j=1}^{\tau} (1-\gamma)|C_j| X_j &< n - (1-\gamma)\left[\sum_{1 \leq j \leq \tau, j \in I^-} |C_j|(1 - e^{-\eta|C_j|}) + Z_\tau^-\right] \\ &< n - (1-\gamma)\left[(1 - e^{-2\eta}) \sum_{1 \leq j \leq \tau, j \in I^-} |C_j| + Z_\tau^-\right] \\ &< n - (1-\gamma)\left[\frac{2n}{1-\gamma} + n\right] < 0 \end{aligned}$$

We have shown that there exists a constant $c' > 0$ such that with probability at least $1 - n^{-d}$ after $c' 2^l n$ edges are sampled by the algorithm at level J all subsequent edges will have their endpoints connected in D_J. Note that we never flipped coins for those edges that did not contain a 2^l-connected component. Setting $c_1 = c' + 1$, we have that w.h.p. $|E \setminus X^J| \leq c_1 2^l n + |E \setminus X^{J-1}|$. By the inductive hypothesis we have that $|E \setminus X^{J-1}| \leq \sum_{1 \leq J' = (l',k') \leq J-2} c_1 2^{l'} n$, which together with the previous estimate gives us the desired result.

It now follows that $|E \setminus X^J| \leq \sum_{1 \leq J' = (l',k') \leq J-1} c_1 2^{l'} n = O(K 2^l n)$ w.h.p., finishing the proof of the lemma.

■